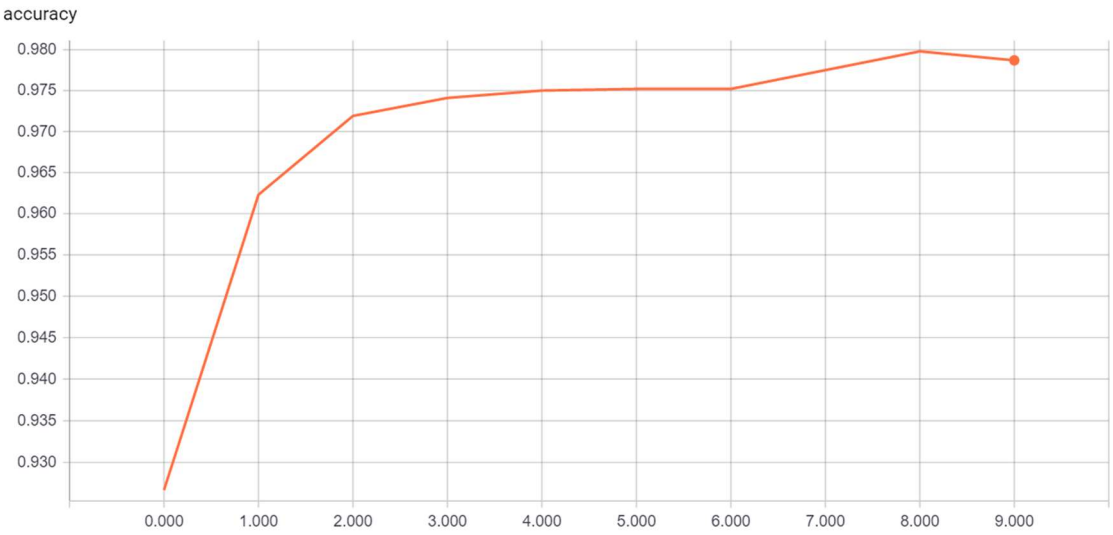

An introduction to neural networks for beginners

By Dr Andy Thomas
Adventures in Machine Learning

Table of Contents

Introduction

Welcome to the "An introduction to neural networks for beginners" book. This book consists of Part A of a much larger, forthcoming book – "From 0 to TensorFlow". The aim of this much larger book is to get you up to speed with all you need to start on the deep learning journey using TensorFlow. What is deep learning, and what is TensorFlow? Deep learning is the field of machine learning that is making many state-of-the-art advancements, from beating players at Go and Poker, to speeding up drug discovery and assisting self-driving cars. If these types of cutting edge applications excite you like they excite me, then you will be interesting in learning as much as you can about deep learning. However, that requires you to know quite a bit about how neural networks work. This will be what this book covers – getting you up to speed on the basic concepts of neural networks and how to create them in Python.

WHO I AM AND MY APPROACH

I am an engineer who works in the energy / utility business who uses machine learning almost daily to excel in my duties. I believe that knowledge of machine learning, and its associated concepts, gives you a significant edge in many different industries, and allows you to approach a multitude of problems in novel and interesting ways. I also maintain an avid interest in machine and deep learning in my spare time, and wish to leverage my previous experience as a university lecturer and academic to educate others in the coming AI and machine learning revolution. My main base for doing this is my website – Adventures in Machine Learning.

Some educators in this area tend to focus solely on the code, with neglect of the theory. Others focus more on the theory, with neglect of the code. There are problems with both these types of approaches. The first leads to a stunted understanding of what one is doing – you get quite good at implementing frameworks but when something goes awry or not quite to plan, you have no idea how to fix it. The second often leads to people getting swamped in theory and mathematics and losing interest before implementing anything in code.

My approach is to try to walk a middle path – with some focus on theory but only as much as is necessary before trying it out in code. I also take things slowly, in a step-by-step

fashion as much as possible. I get frustrated when educators take multiple steps at once and perform large leaps in logic, which makes things difficult to follow, so I assume my readers are likewise annoyed at such leaps and therefore I try not to assume too much.

THE CODE, PRE-REQUISITES AND INSTALLATION

This book will feature snippets of code as we go through the explanations, however the full set of code can be found for download at my github repository. This book does require some loose pre-requisites of the reader – these are as follows:

- A basic understanding of Python variables, arrays, functions, loops and control statements
- A basic understanding of the numpy library, and multi-dimensional indexing
- Basic matrix multiplication concepts and differentiation

While I list these points as pre-requisites, I expect that you will still be able to follow along reasonably well if you are lacking in some of these areas. I expect you'll be able to pick up these ideas as you go along – I'll provide links and go slowly to ensure that is the case.

To install the required software, consult the following links:

- Python 3.5 (this version is required for TensorFlow): https://www.python.org/downloads/
- Numpy: https://www.scipy.org/install.html
- Sci-kit learn: http://scikit-learn.org/stable/install.html

It may be easier for you to install Anaconda, which comes with most of these packages ready to go and allows easy installation of virtual environments.

Part 1 – Introduction to neural networks

1.1 WHAT ARE ARTIFICIAL NEURAL NETWORKS?

Artificial neural networks (ANNs) are software implementations of the neuronal structure of our brains. We don't need to talk about the complex biology of our brain structures, but suffice to say, the brain contains neurons which are kind of like organic switches. These can change their output state depending on the strength of their electrical or chemical input. The neural network in a person's brain is a hugely interconnected network of neurons, where the output of any given neuron may be the input to thousands of other neurons. Learning occurs by repeatedly activating certain neural connections over others, and this reinforces those connections. This makes them more likely to produce a desired outcome given a specified input. This learning involves feedback – when the desired outcome occurs, the neural connections causing that outcome becomes strengthened.

Artificial neural networks attempt to simplify and mimic this brain behavior. They can be trained in a supervised or unsupervised manner. In a supervised ANN, the network is trained by providing matched input and output data samples, with the intention of getting the ANN to provide a desired output for a given input. An example is an e-mail spam filter – the input training data could be the count of various words in the body of the e-mail, and the output training data would be a classification of whether the e-mail was truly spam or not. If many examples of e-mails are passed through the neural network this allows the network to learn what input data makes it likely that an e-mail is spam or not. This learning takes place be adjusting the weights of the ANN connections, but this will be discussed further in the next section.

Unsupervised learning in an ANN is an attempt to get the ANN to "understand" the structure of the provided input data "on its own". This type of ANN will not be discussed in this book.

1.2 THE STRUCTURE OF AN ANN

1.2.1 The artificial neuron

The biological neuron is simulated in an ANN by an activation function. In classification tasks (e.g. identifying spam e-mails) this activation function must have a "switch on" characteristic – in other words, once the input is greater than a certain value, the output should change state i.e. from 0 to 1, from -1 to 1 or from 0 to >0. This simulates the "turning on" of a biological neuron. A common activation function that is used is the sigmoid function:

$$f(z) = \frac{1}{1 + exp(-z)}$$

Which looks like this:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

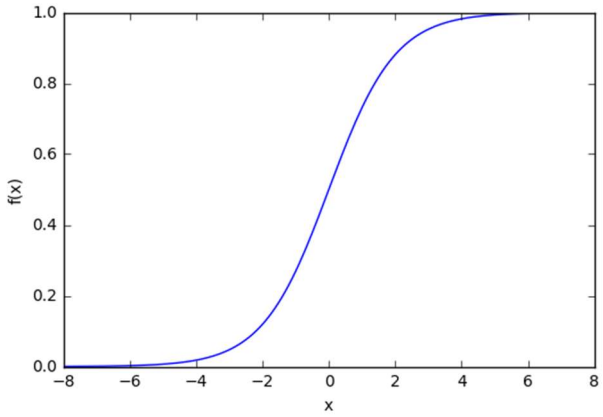

Figure 1 The sigmoid function

As can be seen in the figure above, the function is "activated" i.e. it moves from 0 to 1 when the input x is greater than a certain value. The sigmoid function isn't a step function however, the edge is "soft", and the output doesn't change instantaneously. This means that there is a derivative of the function and this is important for the training algorithm which is discussed more in Section 1.4.5 Backpropagation in depth.

1.2.2 Nodes

As mentioned previously, biological neurons are connected hierarchical networks, with the outputs of some neurons being the inputs to others. We can represent these networks as connected layers of nodes. Each node takes multiple weighted inputs, applies the activation function to the summation of these inputs, and in doing so generates an output. I'll break this down further, but to help things along, consider the diagram below:

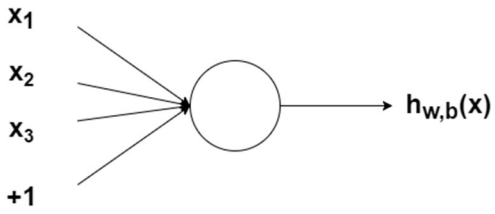

Figure 2 Node with inputs

The circle in the image above represents the node. The node is the "seat" of the activation function, and takes the weighted inputs, sums them, then inputs them to the activation function. The output of the activation function is shown as *h* in the above diagram. Note: a *node* as I have shown above is also called a *perceptron* in some literature.

What about this "weight" idea that has been mentioned? The weights are real valued numbers (i.e. not binary 1s or 0s), which are multiplied by the inputs and then summed up in the node. So, in other words, the weighted input to the node above would be:

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

Here the w_i values are weights (ignore the $\boldsymbol{b}$ for the moment). What are these weights all about? Well, they are the variables that are changed during the learning process, and, along with the input, determine the output of the node. The $\boldsymbol{b}$ is the weight of the +1 *bias* element – the inclusion of this bias enhances the flexibility of the node, which is best demonstrated in an example.

1.2.3 The bias

Let's take an extremely simple node, with only one input and one output:

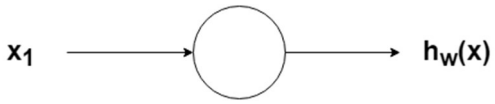

Figure 3 Simple node

The input to the activation function of the node in this case is simply $x_1 w_1$. What does changing w_1 do in this simple network?

```
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w = 0.5'
l2 = 'w = 1.0'
l3 = 'w = 2.0'
for w, l in [(w1, l1), (w2, l2), (w3, l3)]:
    f = 1 / (1 + np.exp(-x*w))
    plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_w(x)')
plt.legend(loc=2)
plt.show()
```

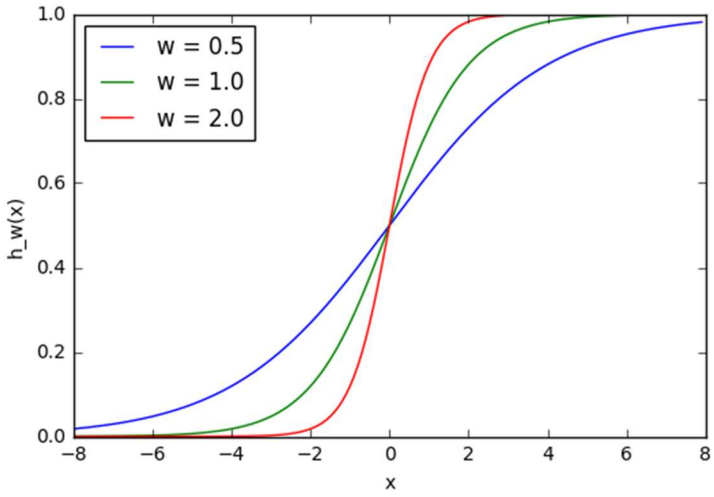

Figure 4 Effect of adjusting weights

Here we can see that changing the weight changes the slope of the output of the sigmoid activation function, which is obviously useful if we want to model different strengths of relationships between the input and output variables. However, what if we only want the output to change when x is greater than 1? This is where the bias comes in – let's consider the same network with a bias input:

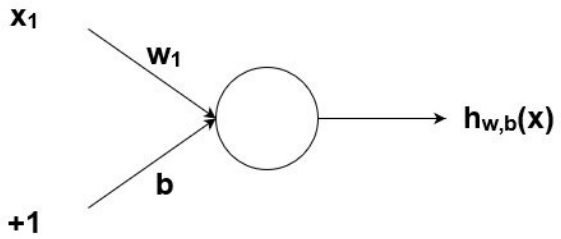

Figure 5 Node with bias

```
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b = -8.0'
l2 = 'b = 0.0'
l3 = 'b = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
    f = 1 / (1 + np.exp(-(x*w+b)))
    plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_wb(x)')
plt.legend(loc=2)
plt.show()
```

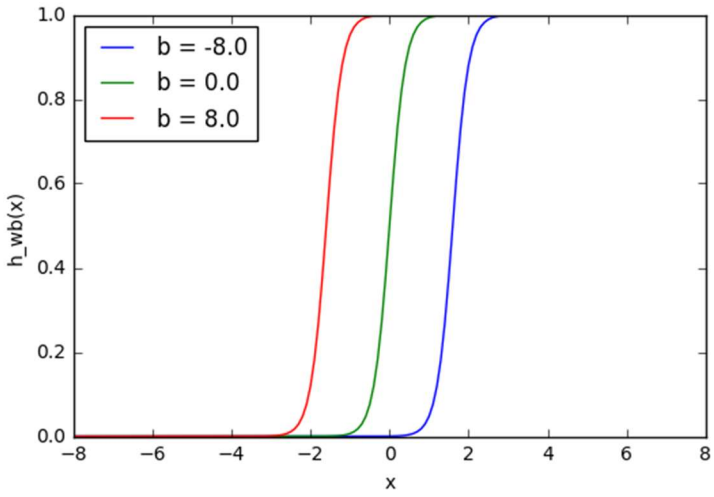

Figure 6 Effect of bias adjustments

In this case, the w_1 has been increased to simulate a more defined "turn on" function. As you can see, by varying the bias "weight" ***b***, you can change when the node activates. Therefore, by adding a bias term, you can make the node simulate a generic **if** function, i.e. *if (x > z) then 1 else 0*. Without a bias term, you are unable to vary the *z* in that if statement, it will be always stuck around 0. This is obviously very useful if you are trying to simulate conditional relationships.

1.2.4 Putting together the structure

Hopefully the previous explanations have given you a good overview of how a given node/neuron/perceptron in a neural network operates. However, as you are probably aware, there are many such interconnected nodes in a fully fledged neural network. These structures can come in a myriad of different forms, but the most common simple neural

network structure consists of an *input layer*, a *hidden layer* and an *output layer*. An example of such a structure can be seen below:

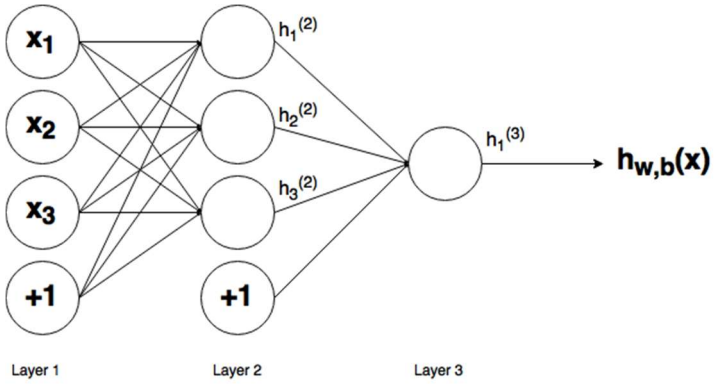

Figure 7 Three layer neural network

The three layers of the network can be seen in the above figure – Layer 1 represents the ***input layer***, where the external input data enters the network. Layer 2 is called the ***hidden layer*** as this layer is not part of the input or output. Note: neural networks can have many hidden layers, but in this case for simplicity I have just included one. Finally, Layer 3 is the ***output layer***. You can observe the many connections between the layers, in particular between Layer 1 (L1) and Layer 2 (L2). As can be seen, each node in L1 has a connection to all the nodes in L2. Likewise for the nodes in L2 to the single output node L3. Each of these connections will have an associated weight.

1.2.5 The notation

The maths below requires some fairly precise notation so that we know what we are talking about. The notation I am using here is similar to that used in the Stanford deep learning tutorial. In the upcoming equations, each of these weights are identified with the following notation: ${w_{ij}}^{(l)}$. i refers to the node number of the connection in layer $l+1$ and j refers to the node number of the connection in layer l. Take special note of this order. So, for the connection between node 1 in layer 1 and node 2 in layer 2, the weight notation would be ${w_{21}}^{(1)}$. This notation may seem a bit odd, as you would expect the i and j to refer the node numbers in layers l and $l+1$ respectively (i.e. in the direction of input to output), rather than the opposite. However, this notation makes more sense when you add the bias.

As you can observe in the figure above – the (+1) bias is connected to each of the nodes in the subsequent layer. The bias in layer 1 is connected to the all the nodes in layer two. Because the bias is not a true node with an activation function, it has no inputs (it always outputs the value +1). The notation of the bias weight is ${b_i}^{(l)}$, where i is the node number in the layer $l+1$ – the same as used for the normal weight notation ${w_{21}}^{(1)}$. So, the

weight on the connection between the bias in layer 1 and the second node in layer 2 is given by ${b_2}^{(1)}$.

Remember, these values – ${w_{ij}}^{(1)}$ and ${b_i}^{(l)}$– all need to be calculated in the training phase of the ANN.

Finally, the node output notation is ${h_j}^{(l)}$, where j denotes the node number in layer l of the network. As can be observed in the three layer network above, the output of node 2 in layer 2 has the notation of ${h_2}^{(2)}$.

Now that we have the notation all sorted out, it is now time to look at how you calculate the output of the network when the input and the weights are known. The process of calculating the output of the neural network given these values is called the *feed-forward* pass or process.

1.3 THE FEED-FORWARD PASS

To demonstrate how to calculate the output from the input in neural networks, let's start with the specific case of the three layer neural network that was presented above. Below it is presented in equation form, then it will be demonstrated with a concrete example and some Python code:

$$\begin{aligned} h_1^{(2)} &= f(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)}) \\ h_2^{(2)} &= f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}) \\ h_3^{(2)} &= f(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)}) \\ h_{W,b}(x) &= h_1^{(3)} = f(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}) \end{aligned}$$

In the equation above $f(\cdot)$ refers to the node activation function, in this case the sigmoid function. The first line, ${h_1}^{(2)}$ is the output of the first node in the second layer, and its inputs are $w_{11}^{(1)}x_1$, $w_{12}^{(1)}x_2$, $w_{13}^{(1)}x_3$ and $b_1^{(1)}$. These inputs can be traced in the three-layer connection diagram above. They are simply summed and then passed through the activation function to calculate the output of the first node. Likewise, for the other two nodes in the second layer.

The final line is the output of the only node in the third and final layer, which is ultimate output of the neural network. As can be observed, rather than taking the weighted input variables (x_1, x_2, x_3), the final node takes as input the weighted output of the nodes of the second layer ($h_1^{(2)}$, $h_2^{(2)}$, $h_3^{(2)}$), plus the weighted bias. Therefore, you can see in equation form the hierarchical nature of artificial neural networks.

1.3.1 A feed-forward example

Now, let's do a simple first example of the output of this neural network in Python. First things first, notice that the weights between layer 1 and 2 ($w_{11}^{(1)}, w_{12}^{(1)}, \ldots$) are ideally suited to matrix representation (check out this link to brush up on matrices)? Observe:

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix}$$

```
import numpy as np
w1 = np.array([[0.2, 0.2, 0.2], [0.4, 0.4, 0.4], [0.6, 0.6, 0.6]])
```

If you're not sure about how numpy arrays work, check out the documentation here. Here I have just filled up the layer 1 weight array with some example weights. We can do the same for the layer 2 weight array:

$$W^{(2)} = (w_{11}^{(2)} \quad w_{12}^{(2)} \quad w_{13}^{(2)})$$

```
w2 = np.zeros((1, 3))
w2[0,:] = np.array([0.5, 0.5, 0.5])
```

We can also setup some dummy values in the layer 1 bias weight array/vector, and the layer 2 bias weight (which is only a single value in this neural network structure – i.e. a scalar):

```
b1 = np.array([0.8, 0.8, 0.8])
b2 = np.array([0.2])
```

Finally, before we write the main program to calculate the output from the neural network, it's handy to setup a separate Python function for the activation function:

```
def f(x):
    return 1 / (1 + np.exp(-x))
```

1.3.2 Our first attempt at a feed-forward function

Below is a simple way of calculating the output of the neural network, using nested loops in python. We'll look at more efficient ways of calculating the output shortly.

```
def simple_looped_nn_calc(n_layers, x, w, b):
    for l in range(n_layers-1):
        #Setup the input array which the weights will be multiplied by for each layer
        #If it's the first layer, the input array will be the x input vector
        #If it's not the first layer, the input to the next layer will be the
        #output of the previous layer
        if l == 0:
            node_in = x
        else:
            node_in = h
        #Setup the output array for the nodes in layer l + 1
        h = np.zeros((w[l].shape[0],))
        #loop through the rows of the weight array
        for i in range(w[l].shape[0]):
            #setup the sum inside the activation function
            f_sum = 0
            #loop through the columns of the weight array
            for j in range(w[l].shape[1]):
                f_sum += w[l][i][j] * node_in[j]
            #add the bias
            f_sum += b[l][i]
            #finally use the activation function to calculate the
            #i-th output i.e. h1, h2, h3
            h[i] = f(f_sum)
    return h
```

This function takes as input the number of layers in the neural network, the x input array/vector, then Python tuples or lists of the weights and bias weights of the network, with each element in the tuple/list representing a layer l in the network. In other words, the inputs are setup in the following:

```
w = [w1, w2]
b = [b1, b2]
#a dummy x input vector
x = [1.5, 2.0, 3.0]
```

The function first checks what the input is to the layer of nodes/weights being considered. If we are looking at the first layer, the input to the second layer nodes is the input vector *x* multiplied by the relevant weights. After the first layer though, the inputs to subsequent layers are the output of the previous layers. Finally, there is a nested loop through the relevant *I* and *j* values of the weight vectors and the bias. The function uses the dimensions of the weights for each layer to figure out the number of nodes and therefore the structure of the network.

Calling the function:

```
simple_looped_nn_calc(3, x, w, b)
```

gives the output of 0.8354. We can confirm this results by manually performing the calculations in the original equations:

$$
\begin{aligned}
h_1^{(2)} &= f(0.2 * 1.5 + 0.2 * 2.0 + 0.2 * 3.0 + 0.8) = 0.8909 \\
h_2^{(2)} &= f(0.4 * 1.5 + 0.4 * 2.0 + 0.4 * 3.0 + 0.8) = 0.9677 \\
h_3^{(2)} &= f(0.6 * 1.5 + 0.6 * 2.0 + 0.6 * 3.0 + 0.8) = 0.9909 \\
h_{W,b}(x) &= h_1^{(3)} = f(0.5 * 0.8909 + 0.5 * 0.9677 + 0.5 * 0.9909 + 0.2) = 0.8354
\end{aligned}
$$

1.3.3 A more efficient implementation

As was stated earlier – using loops isn't the most efficient way of calculating the feed forward step in Python. This is because the loops in Python are notoriously slow. An alternative, more efficient mechanism of doing the feed forward step in Python and numpy will be discussed shortly. We can benchmark how efficient the algorithm is by using the *%timeit* function in IPython, which runs the function a number of times and returns the average time that the function takes to run:

```
%timeit simple_looped_nn_calc(3, x, w, b)
```

Running this tells us that the looped feed forward takes 40μs. A result in the tens of microseconds sounds very fast, but when applied to very large practical NNs with 100s of nodes per layer, this speed will become prohibitive, especially when training the network, as will become clear later in this tutorial. If we try a four layer neural network using the same code, we get significantly worse performance – 70μs in fact.

1.3.4 Vectorisation in neural networks

There is a way to write the equations even more compactly, and to calculate the feed forward process in neural networks more efficiently, from a computational perspective. Firstly, we can introduce a new variable $z_i^{(l)}$ which is the summated input into node *I* of layer *l*, including the bias term. So in the case of the first node in layer 2, *z* is equal to:

$$z_1^{(2)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)} = \sum_{j=1}^{n} w_{ij}^{(1)} x_i + b_i^{(1)}$$

where n is the number of nodes in layer 1. Using this notation, the unwieldy previous set of equations for the example three layer network can be reduced to:

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
h^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}h^{(2)} + b^{(2)} \\
h_{W,b}(x) &= h^{(3)} = f(z^{(3)})
\end{aligned}
$$

Note the use of capital W to denote the matrix form of the weights. It should be noted that all of the elements in the above equation are now matrices / vectors. If you're unfamiliar with these concepts, they will be explained more fully in the next section. Can the above equation be simplified even further? Yes, it can. We can forward propagate the calculations through any number of layers in the neural network by generalising:

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}h^{(l)} + b^{(l)} \\
h^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

Here we can see the general feed forward process, where the output of layer l becomes the input to layer $l + 1$. We know that $h^{(1)}$ is simply the input layer x and $h^{(n_l)}$ (where n_l is the number of layers in the network) is the output of the output layer. Notice in the above equations that we have dropped references to the node numbers i and j – how can we do this? Don't we still have to loop through and calculate all the various node inputs and outputs?

The answer is that we can use matrix multiplications to do this more simply. This process is called "vectorisation" and it has two benefits – first, it makes the code less complicated, as you will see shortly. Second, we can use fast linear algebra routines in Python (and other languages) rather than using loops, which will speed up our programs. Numpy can handle these calculations easily. First, for those who aren't familiar with matrix operations, the next section is a brief recap.

1.3.5 Matrix multiplication

Let's expand out $z^{(l+1)} = W^{(l)}h^{(l)} + b^{(l)}$ in explicit matrix/vector form for the input layer (i.e. $h^{(l)} = x$):

$$
\begin{aligned}
z^{(2)} &= \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)} \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)} \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)} \end{pmatrix}
\end{aligned}
$$

For those who aren't aware of how matrix multiplication works, it is a good idea to scrub up on matrix operations. There are many sites which cover this well. However, just quickly, when the weight matrix is multiplied by the input layer vector, each element in the *row* of the weight matrix is multiplied by each element in the single *column* of the input vector, then summed to create a new (3 x 1) vector. Then you can simply add the bias weights vector to achieve the final result.

You can observe how each row of the final result above corresponds to the argument of the activation function in the original non-matrix set of equations above. If the activation function is capable of being applied element-wise (i.e. to each row separately in the $z^{(1)}$ vector), then we can do all our calculations using matrices and vectors rather than slow Python loops. Thankfully, numpy allows us to do just that, with reasonably fast matrix operations and element-wise functions. Let's have a look at a much more simplified (and faster) version of the simple_looped_nn_calc:

```
def matrix_feed_forward_calc(n_layers, x, w, b):
    for l in range(n_layers-1):
        if l == 0:
            node_in = x
        else:
            node_in = h
        z = w[l].dot(node_in) + b[l]
        h = f(z)
    return h
```

Note line 7 where the matrix multiplication occurs – if you just use the ** symbol when multiplying the weights by the node input vector in numpy it will attempt to perform some sort of element-wise multiplication, rather than the true matrix multiplication that we desire. Therefore you need to use the *a.dot(b)* notation when performing matrix multiplication in numpy.

If we perform *%timeit* again using this new function and a simple 4 layer network, we only get an improvement of 24*µs* (a reduction from 70*µs* to 46*µs*). However, if we increase the size of the 4 layer network to layers of 100-100-50-10 nodes the results are much more impressive. The Python looped based method takes a whopping 41*ms* – note, that is milliseconds, and the vectorised implementation only takes 84*µs* to forward propagate through the neural network. By using vectorised calculations instead of Python loops we have increased the efficiency of the calculation 500 fold! That's a huge improvement. There is even the possibility of faster implementations of matrix operations using deep learning packages such as TensorFlow and Theano which utilise your computer's GPU (rather than the CPU), the architecture of which is more suited to fast matrix computations (I have a TensorFlow tutorial post also)

That brings us to an end of the feed-forward introduction for neural networks. The next section will deal with how to train a neural network so that it can perform classification tasks, using gradient descent and backpropagation.

1.4 GRADIENT DESCENT AND OPTIMISATION

As mentioned in Section 1.2.2 Nodes, the setting of the values of the weights which link the layers in the network is what constitutes the training of the system. In supervised learning, the idea is to reduce the *error* between the input and the desired output. So if we have a neural network with one output layer, and given some input x we want the neural network to output a 2, yet the network actually produces a 5, a simple expression of the *error* is $abs(2-5)=3$.For the mathematically minded, this would be the L^1 norm of the error (don't worry about it if you don't know what this is).

The idea of supervised learning is to provide many input-output pairs of known data and vary the weights based on these samples so that the error expression is minimised. We can specify these input-output pairs as $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ where m is the number of training samples that we have on hand to train the weights of the network. Each of these inputs or outputs can be vectors – that is $x^{(1)}$ is not necessarily just one value, it could be an N dimensional series of values. For instance, let's say that we're training a spam-detection neural network – in such a case $x^{(1)}$ could be a count of all the different significant words in an e-mail e.g.:

$$x^{(1)} = \begin{pmatrix} No.\,of\text{“}prince\text{”} \\ No.\,of\text{“}nigeria\text{”} \\ No.\,of\text{“}extension\text{”} \\ \vdots \\ No.\,of\text{“}mum\text{”} \\ No.\,of\text{“}burger\text{”} \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

$y^{(1)}$ in this case could be a single scalar value, either a 1 or a 0 to designate whether the e-mail is spam or not. Or, in other applications it could be a K dimensional vector. As an example, say we have input x that is a vector of the pixel greyscale readings of an image. We also have an output y that is a 26 dimensional vector that designates, with a 1 or 0, what letter of the alphabet is shown in the image i.e (1,0, … ,0) for a, (0,1, … ,0) for b and so on. This 26 dimensional output vector could be used to classify letters in photographs.

In training the network with these *(x,y)* pairs, the goal is to get the neural network better and better at predicting the correct *y* given *x*. This is performed by varying the weights so as to minimize the error. How do we know how to vary the weights, given an error in the output of the network? This is where the concept of ***gradient descent*** comes in handy. Consider the diagram below:

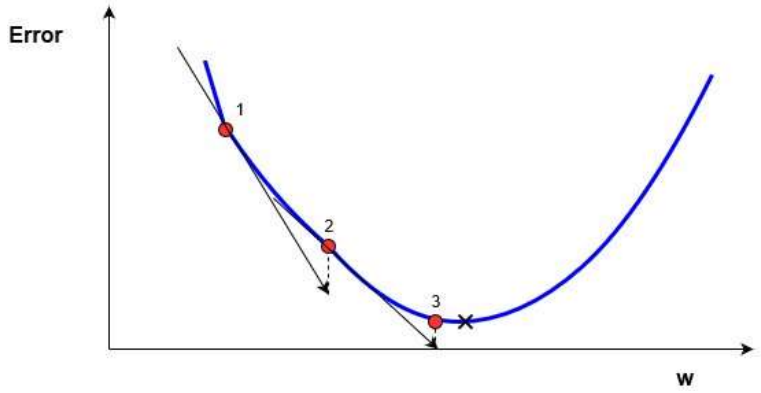

Figure 8 Simple, one-dimensional gradient descent

In this diagram we have a blue plot of the error depending on a single scalar weight value, *w*. The minimum possible error is marked by the black cross, but we don't know what *w* value gives that minimum error. We start out at a random value of *w*, which gives an error marked by the red dot on the curve labelled with "1". We need to change *w* in a way to approach that minimum possible error, the black cross. One of the most common ways of approaching that value is called ***gradient descent***.

To proceed with this method, first the gradient of the error with respect to *w* is calculated at point "1". For those who don't know, the *gradient* is the slope of the error curve at that point. It is shown in the diagram above by the black arrow which "pierces" point "1". The gradient also gives directional information – if it is positive with respect to an increase in *w*, a step in that *direction* will lead to an increase in the error. If it is negative with respect to an *increase* in *w* (as it is in the diagram above), a step in that will lead to a *decrease* in the error. Obviously, we wish to make a step in *w* that will lead to a decrease in the error. The magnitude of the *gradient* or the "steepness" of the slope, gives an indication of how fast the error curve or function is changing at that point. The higher the magnitude of the gradient, the faster the error is changing at that point with respect to *w*.

The gradient descent method uses the gradient to make an informed step change in *w* to lead it towards the minimum of the error curve. This is an *iterative* method, that involves multiple steps. Each time, the *w* value is updated according to:

$$w_{new} = w_{old} - \alpha * \nabla error$$

Here w_{new} denotes the new *w* position, w_{old} denotes the current or old *w* position, $\nabla error$ is the gradient of the error at w_{old} and α is the step size. The step size α will determine how

quickly the solution converges on the minimum error. However, this parameter has to be tuned – if it is too large, you can imagine the solution bouncing around on either side of the minimum in the above diagram. This will result in an optimisation of *w* that does not converge. As this iterative algorithm approaches the minimum, the gradient or change in the error with each step will reduce. You can see in the graph above that the gradient lines will "flatten out" as the solution point approaches the minimum. As the solution approaches the minimum error, because of the decreasing gradient, it will result in only small improvements to the error. When the solution approaches this "flattening" out of the error we want to exit the iterative process. This exit can be performed by either stopping after a certain number of iterations or via some sort of "stop condition". This stop condition might be when the change in the error drops below a certain limit, often called the *precision*.

1.4.1 A simple example in code

Below is an example of a simple Python implementation of gradient descent for solving the minimum of the equation $f(x) = x^4 - 3x^3 + 2$ taken from Wikipedia. The gradient of this function is able to be calculated analytically (i.e. we can do it easily using calculus, which we can't do with many real world applications) and is $f'(x) = 4x^3 - 9x^2$. This means at every value of x, we can calculate the gradient of the function by using a simple equation. Again, using calculus we can know that the exact minimum of this equation is $x = 2.25$.

```
x_old = 0 # The value does not matter as long as abs(x_new - x_old) > precision
x_new = 6 # The algorithm starts at x=6
gamma = 0.01 # step size
precision = 0.00001

def df(x):
    y = 4 * x**3 - 9 * x**2
    return y

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new += -gamma * df(x_old)

print("The local minimum occurs at %f" % x_new)
```

This function prints "The local minimum occurs at 2.249965", which agrees with the exact solution within the precision. This code implements the weight adjustment algorithm that I showed above, and can be seen to find the minimum of the function correctly within the given precision. This is a very simple example of gradient descent, and finding the gradient works quite differently when training neural networks. However, the main idea remains – we figure out the gradient of the neural network then adjust the weights in

a step to try to get closer to the minimum error that we are trying to find. Another difference between this toy example of gradient descent is that the weight vector is multi-dimensional, and therefore the gradient descent method must search a multi-dimensional space for the minimum point.

The way we figure out the gradient of a neural network is via the famous ***backpropagation*** method, which will be discussed shortly. First however, we must look at the error function more closely.

1.4.2 The cost function

Previously, we've talked about iteratively minimising the error of the output of the neural network by varying the weights in gradient descent. However, as it turns out, there is a mathematically more generalised way of looking at things that allows us to reduce the error while also preventing things like *overfitting* (this will be discussed more in later articles). This more general optimisation formulation revolves around minimising what's called the ***cost function***. The equivalent cost function of a single training pair (x^z, y^z) in a neural network is:

$$\begin{aligned} J(w,b,x,y) &= \frac{1}{2} \| \, y^z - h^{(n_l)}(x^z) \, \|^2 \\ &= \frac{1}{2} \| \, y^z - y_{pred}(x^z) \, \|^2 \end{aligned}$$

This shows the cost function of the z_{th} training sample, where $h^{(n_l)}$ is the output of the final layer of the neural network i.e. the output of the neural network. I've also represented $h^{(n_l)}$ as y_{pred} to highlight the prediction of the neural network given x^z. The two vertical lines represent the L^2 norm of the error, or what is known as the sum-of-squares error (SSE). SSE is a very common way of representing the error of a machine learning system. Instead of taking just the absolute error $abs(y_{pred}(x^z) - y^z)$, we use the square of the error. There are many reasons why the SSE is often used which will not be discussed here – suffice to say that this is a very common way of representing the errors in machine learning. The $\frac{1}{2}$ out the front is just a constant added that tidies things up when we differentiate the cost function, which we'll be doing when we perform backpropagation.

Note that the formulation for the cost function above is for a single (x, y) training pair. We want to minimise the cost function over all our *m* training pairs. Therefore, we want to find the minimum *mean squared error* (MSE) over all the training samples:

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2}\| y^z - h^{(n_l)}(x^z) \|^2$$
$$= \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

So, how do you use the cost function J above to train the weights of our network? Using gradient descent and backpropagation. First, let's look at gradient descent more closely in neural networks.

1.4.3 Gradient descent in neural networks

Gradient descent for every weight $w_{(ij)}^{(l)}$ and every bias $b_i^{(l)}$ in the neural network looks like the following:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha\frac{\partial}{\partial w_{ij}^{(l)}}J(w,b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(w,b)$$

Basically, the equation above is like the previously shown gradient descent algorithm: $w_{new} = w_{old} - \alpha * \nabla error$. The new and old subscripts are missing, but the values on the left side of the equation are *new* and the values on the right side are *old*. Again, we have an iterative process whereby the weights are updated in each iteration, this time based on the cost function $J(w,b)$.

The values $\frac{\partial}{\partial w_{ij}^{(l)}}$ and $\frac{\partial}{\partial b_i^{(l)}}$ are the *partial derivatives* of the single sample cost function based on the weight values. What does this mean? Recall that for the simple gradient descent example mentioned previously, each step depends on the *slope* of the error/cost term with respect to the weights. Another word for slope or gradient is the *derivative*. A normal derivative has the notation $\frac{d}{dx}$. If x in this instance is a vector, then such a derivative will also be a vector, displaying the gradient in all the dimensions of x.

1.4.4 A two dimensional gradient descent example

Let's take the example of a standard two-dimensional gradient descent problem. Below is a diagram of an iterative two-dimensional gradient descent run:

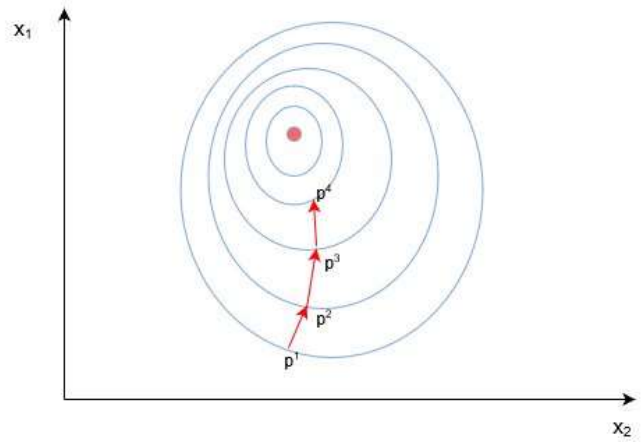

Figure 9 Two-dimensional gradient descent

The blue lines in the above diagram are the contour lines of the cost function – designating regions with an error value that is approximately the same. As can be observed in the diagram above, each step ($p_1 \rightarrow p_2 \rightarrow p_3$) in the gradient descent involves a gradient or derivative that is an arrow/vector. This vector spans both the [$x1$,$x2$] dimensions, as the solution works its way towards the minimum in the centre. So, for instance, the derivative evaluated at p_1 might be $\frac{d}{dx} = [2.1,0.7]$, where the derivative is a vector to designate the two directions. The *partial* derivative $\frac{\partial}{\partial x_1}$ in this case would be a scalar → [2.1] – in other words, it is the gradient in only one direction of the search space (x_1). In gradient descent, it is often the case that the partial derivative of all the possible search directions are calculated, then "gathered up" to determine a new, complete, step direction.

In neural networks, we don't have a simple cost function where we can easily evaluate the gradient, like we did in our toy gradient descent example ($f(x) = x^4 - 3x^3 + 2$). In fact, things are even trickier. While we can compare the output of the neural network to our expected training value, y(z) and feasibly look at how changing the weights of the output layer would change the cost function for the sample (i.e. calculating the gradient), how on earth do we do that for all the *hidden* layers of the network?

The answer to that is the backpropagation method. This method allows us to "share" the cost function or error to all the weights in the network – or in other words, it allows us to determine how much of the error is caused by any given weight.

1.4.5 Backpropagation in depth

In this section, I'm going to delve into the maths a little. If you're wary of the maths of how backpropagation works, then it may be best to skip this section. Section 1.5 Implementing the neural network in Python will show you how to implement backpropagation in code – so if you want to skip straight on to using this method, feel free

to skip the rest of this section. However, if you don't mind a little bit of maths, I encourage you to push on to the end of this section as it will give you a good depth of understanding in training neural networks. This will be invaluable to understanding some of the key ideas in deep learning, such as the vanishing gradient problem, rather than just being a code cruncher who doesn't really understand how the code works.

First let's recall some of the foundational equations from Section 1.3 The feed-forward pass for the following three layer neural network:

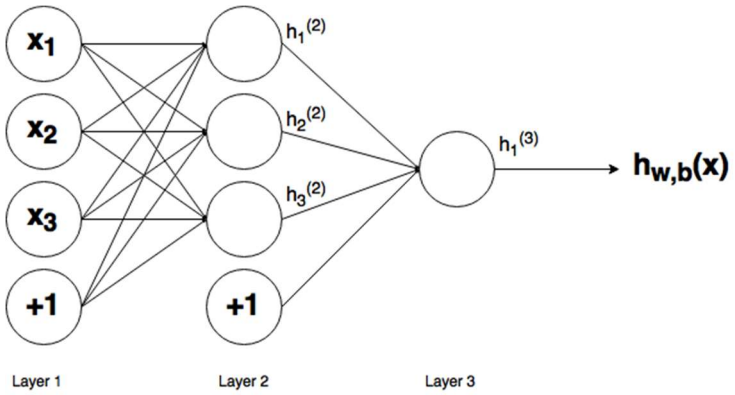

Figure 10 Three layer neural network (again)

The output of this neural network can be calculated by:

$$h_{W,b}(x) = h_1^{(3)} = f(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)})$$

We can also simplify the above to $h_1^{(3)} = f(z_1^{(2)})$ by defining $z_1^{(2)}$ as:

$$z_1^{(2)} = w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}$$

Let's say we want to find out how much a change in the weight $w_{12}^{(2)}$ has on the cost function *J*. This is to evaluate $\frac{\partial J}{\partial w_{12}^{(2)}}$. To do so, we must use something called the chain function:

$$\frac{\partial J}{\partial w_{12}^{(2)}} = \frac{\partial J}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$$

If you look at the terms on the right – the numerators "cancel out" the denominators, in the same way that $\frac{2}{5}\frac{5}{2} = \frac{2}{2} = 1$. Therefore we can construct $\frac{\partial J}{\partial w_{12}^{(2)}}$ by stringing together a few partial derivatives (which are quite easy, thankfully). Let's start with $\frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$:

$$\frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} = \frac{\partial}{\partial w_{12}^{(2)}}(w_{11}^{(1)}h_1^{(2)} + w_{12}^{(1)}h_2^{(2)} + w_{13}^{(1)}h_3^{(2)} + b_1^{(1)})$$
$$= \frac{\partial}{\partial w_{12}^{(2)}}(w_{12}^{(1)}h_2^{(2)})$$
$$= h_2^{(2)}$$

The partial derivative of $z_1^{(2)}$ with respect $w_{12}^{(2)}$ only operates on one term within the parentheses, $w_{12}^{(1)}h_2^{(2)}$, as all the other terms don't vary at all when $w_{12}^{(2)}$ does. The derivative of a constant is 1, therefore $\frac{\partial}{\partial w_{12}^{(2)}}(w_{12}^{(1)}h_2^{(2)})$ collapses to just $h_2^{(2)}$, which is simply the output of the second node in layer 2.

The next partial derivative in the chain is $\frac{\partial h_1^{(3)}}{\partial z_1^{(2)}}$, which is the partial derivative of the activation function of the $h_1^{(3)}$ output node. Because of the requirement to be able to derive this derivative, the activation functions in neural networks need to be *differentiable*. For the common sigmoid activation function (shown in Section 1.2.1 The artificial neuron), the derivative is:

$$\frac{\partial h}{\partial z} = f'(z) = f(z)\big(1 - f(z)\big)$$

Where $f(z)$ is the activation function. So far so good – now we have to work out how to deal with the first term $\frac{\partial J}{\partial h_1^{(3)}}$. Remember that $J(w, b, x, y)$ is the mean squared error loss function, which looks like (for our case):

$$J(w, b, x, y) = \frac{1}{2} \parallel y_1 - h_1^{(3)}(z_1^{(2)}) \parallel^2$$

Here y_1 is the training target for the output node. Again using the chain rule:

$$\textbf{Let } u = \parallel y_1 - h_1^{(3)}(z_1^{(2)}) \parallel \textbf{ and } J = \frac{1}{2}u^2$$
$$\textbf{Using } \frac{\partial J}{\partial h} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial h} \textbf{ :}$$
$$\frac{\partial J}{\partial h} = -(y_1 - h_1^{(3)})$$

So we've now figured out how to calculate $\frac{\partial J}{\partial w_{12}^{(2)}}$, at least for the weights connecting the output layer. Before we move to any hidden layers (i.e. layer 2 in our example case), let's introduce some simplifications to tighten up our notation and introduce δ:

$$\delta_i^{(n_l)} = -(y_i - h_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

Where i is the node number of the output layer. In our selected example there is only one such layer, therefore $i = 1$ always in this case. Now we can write the complete cost function derivative as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b, x, y) \quad = h_j^{(l)} \delta_i^{(l+1)}$$

Where, for the output layer in our case, l=2 and i remains the node number.

1.4.6 Propagating into the hidden layers

What about for weights feeding into any hidden layers (layer 2 in our case)? For the weights connecting the output layer, the $\frac{\partial J}{\partial h} = -(y_i - h_i^{(n_l)})$ derivative made sense, as the cost function can be directly calculated by comparing the output layer to the training data. The output of the hidden nodes, however, have no such direct reference, rather, they are connected to the cost function only through mediating weights and potentially other layers of nodes. How can we find the variation in the cost function from changes to weights embedded deep within the neural network? As mentioned previously, we use the *backpropagation* method.

Now that we've done the demanding work using the chain rule, we'll now take a more graphical approach. The term that needs to propagate back through the network is the $\delta_i^{(n_l)}$ term, as this is the network's ultimate connection to the cost function. What about node j in the second layer (hidden layer)? How does it contribute to $\delta_i^{(n_l)}$ in our test network? It contributes via the weight $w_{ij}^{(2)}$ – see the diagram below for the case of j=1 and i=1.

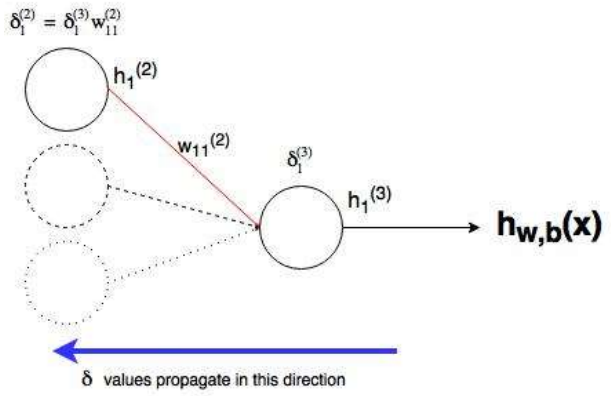

Figure 11 Simple backpropagation illustration

As can be observed from above, the output layer δ is *communicated* to the hidden node by the weight of the connection. In the case where there is only one output layer node, the generalised hidden layer δ is defined as:

$$\delta_j^{(l)} = \delta_1^{(l+1)} w_{1j}^{(l)} f'(z_j)^{(l)}$$

Where *j* is the node number in layer *l*. What about the case where there are multiple output nodes? In this case, the weighted sum of all the communicated errors are taken to calculate $\delta_j^{(l)}$, as shown in the diagram below:

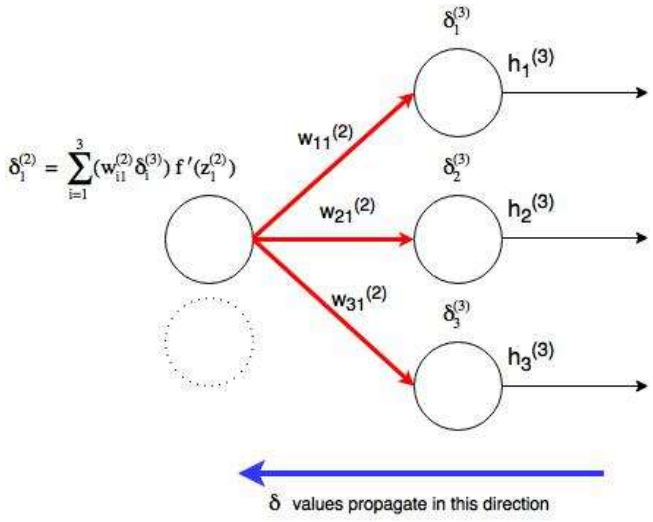

Figure 12 Backpropagation illustration with multiple outputs

As can be observed from the above, each δ value from the output layer is included in the sum used to calculate $\delta_1^{(2)}$, but each output δ is weighted according to the appropriate $w_{i1}^{(2)}$ value. In other words, node 1 in layer 2 contributes to the error of three output nodes, therefore the measured error (or cost function value) at each of these nodes must be "passed back" to the δ value for this node. Now we can develop a generalised expression for the δ values for nodes in the hidden layers:

$$\delta_j^{(l)} = (\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)} \delta_i^{(l+1)}) f'(z_j^{(l)})$$

Where *j* is the node number in layer *l* and *i* is the node number in layer $l + 1$ (which is the same notation we have used from the start). The value $s_{(l+1)}$ is the number of nodes in layer (l+1).

So we now know how to calculate:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b, x, y) = h_j^{(l)} \delta_i^{(l+1)}$$

as shown previously. What about the bias weights? I'm not going to derive them as I did with the normal weights in the interest of saving time / space. However, the reader shouldn't have too many issues following the same steps, using the chain rule, to arrive at:

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b, x, y) = \delta_i^{(l+1)}$$

Great – so we now know how to perform our original gradient descent problem for neural networks:

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w, b) \\ b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(w, b) \end{aligned}$$

However, to perform this gradient descent training of the weights, we would have to resort to loops within loops. As previously shown in Section 1.3.4 Vectorisation in neural networksof this neural network tutorial, performing such calculations in Python using loops is slow for large networks. Therefore, we need to figure out how to vectorise such calculations, which the next section will show.

1.4.7 Vectorisation of backpropagation

To consider how to vectorise the gradient descent calculations in neural networks, let's first look at a naïve vectorised version of the gradient of the cost function (**warning**: this is not in a correct form yet!):

$$\begin{aligned} \frac{\partial J}{\partial W^{(l)}} &= h^{(l)} \delta^{(l+1)} \\ \frac{\partial J}{\partial b^{(l)}} &= \delta^{(l+1)} \end{aligned}$$

Now, let's look at what element of the above equations. What does $h^{(l)}$ look like? Pretty simple, just a $(s_l \times 1)$ vector, where s_l is the number of nodes in layer l. What does the multiplication of $h^{(l)}\delta^{(l+1)}$ look like? Well, because we know that $\alpha \times \frac{\partial J}{\partial W^{(l)}}$ must be the same size of the weight matrix $W^{(l)}$, we know that the outcome of $h^{(l)}\delta^{(l+1)}$ must also be the same size as the weight matrix for layer l. In other words, it has to be of size $(s_{l+1} \times s_l)$.

We know that $\delta^{(l+1)}$ has the dimension $(s_{l+1} \times 1)$ and that $h^{(l)}$ has the dimension of $(s_l \times 1)$. The rules of matrix multiplication show that a matrix of dimension $(\mathbf{n} \times m)$ multiplied by a matrix of dimension $(o \times \mathbf{p})$ will have a product matrix of size $(\mathbf{n} \times \mathbf{p})$. If we perform a straight multiplication between $h^{(l)}$ and $\delta^{(l+1)}$, the number of columns of the first vector (i.e. 1 column) will not equal the number of rows of the second vector (i.e. 3 rows), therefore we can't perform a proper matrix multiplication. The only way we can get

a proper outcome of size$(s_{l+1}\times s_l)$ is by using a matrix ***transpose***. A transpose swaps the dimensions of a matrix around e.g. a $(s_l\times 1)$ sized vector becomes a$(1\times s_l)$ sized vector, and is denoted by a superscript of ***T***. Therefore, we can do the following:

$$\delta^{(l+1)}(h^{(l)})^T = (s_{l+1}\times 1)\times(1\times s_l) = (s_{l+1}\times s_l)$$

As can be observed below, by using the transpose operation we can arrive at the outcome we desired.

A final vectorisation that can be performed is during the weighted addition of the errors in the backpropagation step:

$$\delta_j^{(l)} = (\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)}\delta_i^{(l+1)})f'(z_j^{(l)}) = ((W^{(l)})^T\delta^{(l+1)})\cdot f'(z^{(l)})$$

The · symbol in the above designates an element-by-element multiplication (called the Hadamard product), not a matrix multiplication. Note that the matrix multiplication $((W^{(l)})^T\delta^{(l+1)})$ performs the necessary summation of the weights and δ values – the reader can check that this is the case.

1.4.8 Implementing the gradient descent step

Now, how do we integrate this new vectorisation into the gradient descent steps of our soon-to-be coded algorithm? First, we have to look again at the overall cost function we are trying to minimise (not just the sample-by-sample cost function shown in the preceding equation):

$$J(w,b) \quad = \frac{1}{m}\sum_{z=0}^{m} J\left(W,b,x^{(z)},y^{(z)}\right)$$

As we can observe, the total cost function is the mean of all the sample-by-sample cost function calculations. Also remember the gradient descent calculation (showing the element-by-element version along with the vectorised version):

$$w_{ij}^{(l)} \qquad = w_{ij}^{(l)} - \alpha\frac{\partial}{\partial w_{ij}^{(l)}}J(w,b)$$

$$W^{(l)} \qquad = W^{(l)} - \alpha\frac{\partial}{\partial W^{(l)}}J(w,b)$$

$$= W^{(l)} - \alpha[\frac{1}{m}\sum_{z=1}^{m}\frac{\partial}{\partial W^{(l)}}J(w,b,x^{(z)},y^{(z)})]$$

So that means as we go along through our training samples or batches, we have to have a term that is summing up the partial derivatives of the individual sample cost function

calculations. This term will gather up all the values for the mean calculation. Let's call this "summing up" term $\Delta W^{(l)}$. Likewise, the equivalent bias term can be called $\Delta b^{(l)}$. Therefore, at each sample iteration of the final training algorithm, we have to perform the following steps:

$$\begin{aligned} \Delta W^{(l)} &= \Delta W^{(l)} + \frac{\partial}{\partial W^{(l)}} J\left(w, b, x^{(z)}, y^{(z)}\right) \\ &= \Delta W^{(l)} + \delta^{(l+1)} (h^{(l)})^T \\ \Delta b^{(l)} &= \Delta b^{(1)} + \delta^{(l+1)} \end{aligned}$$

By performing the above operations at each iteration, we slowly build up the previously mentioned sum $\sum_{z=1}^{m} \frac{\partial}{\partial W^{(l)}} J(w, b, x^{(z)}, y^{(z)})$ (and the same for b). Once all the samples have been iterated through, and the Δ values have been summed up, we update the weight parameters:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)}\right] \\ b^{(l)} &= b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)}\right] \end{aligned}$$

1.4.9 The final gradient descent algorithm

So, no we've finally made it to the point where we can specify the entire backpropagation-based gradient descent training of our neural networks. It has taken quite a few steps to show, but hopefully it has been instructive. The final backpropagation algorithm is as follows:

Randomly initialise the weights for each layer $W^{(l)}$
While iterations < iteration limit:
1. Set ΔW and Δb to zero
2. For samples 1 to m:
a. Perform a feed forward pass through all the n_l layers. Store the activation function outputs $h^{(l)}$
b. Calculate the $\delta^{(n_l)}$ value for the output layer
c. Use backpropagation to calculate the $\delta^{(l)}$ values for layers 2 to $n_l - 1$
d. Update the $\Delta W^{(l)}$ and $\Delta b^{(l)}$ each layer
3. Perform a gradient descent step using:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)}\right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)}\right]$$

As specified in the algorithm above, we would repeat the gradient descent routine until we are happy that the average cost function has reached a minimum. At this point, our network is trained and (ideally) ready for use.

The next part of this neural networks tutorial will show how to implement this algorithm to train a neural network that recognises hand-written digits.

1.5 IMPLEMENTING THE NEURAL NETWORK IN PYTHON

In the last section we looked at the theory surrounding gradient descent training in neural networks and the backpropagation method. In this article, we are going to apply that theory to develop some code to perform training and prediction on the MNIST dataset. The MNIST dataset is a kind of go-to dataset in neural network and deep learning examples, so we'll stick with it here too. What it consists of is a record of images of hand-written digits with associated labels that tell us what the digit is. Each image is 8 x 8 pixels in size, and the image data sample is represented by 64 data points which denote the pixel intensity. In this example, we'll be using the MNIST dataset provided in the Python Machine Learning library called scikit learn. An example of the image (and the extraction of the data from the scikit learn dataset) is shown in the code below (for an image of 1):

```
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
import matplotlib.pyplot as plt
plt.gray()
plt.matshow(digits.images[1])
plt.show()
```

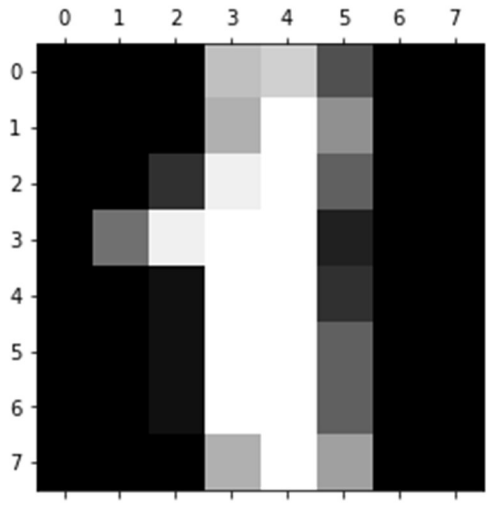

Figure 13 MNIST digit "1"

The code above prints (1797, 64) to show the shape of input data matrix and the pixelated digit "1" in the image above. The code we are going to write in this neural networks tutorial will try and estimate the digits that these pixels represent (using neural networks of course). First things first, we need to get the input data in shape. To do so, we need to do two things:

1. Scale the data
2. Split the data into test and train sets

1.5.1 Scaling data

Why do we need to scale the input data? First, have a look at one of the dataset pixel representations:

```
digits.data[0,:]
Out[2]:
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Notice that the input data ranges from 0 up to 15? It's common practice to scale the input data so that it all fits mostly between either 0 to 1 or with a small range centred around 0 i.e. -1 to 1. Why? Well, it can help the convergence of the neural network and is especially important if we are combining different data types. Thankfully, this is easily done using sci-kit learn:

```
from sklearn.preprocessing import StandardScaler
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
X[0,:]
Out[3]:
array([ 0.        , -0.33501649, -0.04308102,  0.27407152, -0.66447751,
       -0.84412939, -0.40972392, -0.12502292, -0.05907756, -0.62400926,
        0.4829745 ,  0.75962245, -0.05842586,  1.12772113,  0.87958306,
       -0.13043338, -0.04462507,  0.11144272,  0.89588044, -0.86066632,
       -1.14964846,  0.51547187,  1.90596347, -0.11422184, -0.03337973,
        0.48648928,  0.46988512, -1.49990136, -1.61406277,  0.07639777,
        1.54181413, -0.04723238,  0.        ,  0.76465553,  0.05263019,
       -1.44763006, -1.73666443,  0.04361588,  1.43955804,  0.        ,
       -0.06134367,  0.8105536 ,  0.63011714, -1.12245711, -1.06623158,
        0.66096475,  0.81845076, -0.08874162, -0.03543326,  0.74211893,
        1.15065212, -0.86867056,  0.11012973,  0.53761116, -0.75743581,
       -0.20978513, -0.02359646, -0.29908135,  0.08671869,  0.20829258,
       -0.36677122, -1.14664746, -0.5056698 , -0.19600752])
```

The scikit learn standard scaler by default normalises the data by subtracting the mean and dividing by the standard deviation. As can be observed, most of the data points are centered around zero and contained within -2 and 2. This is a good starting point. There is no real need to scale the output data *y*.

1.5.2 Creating test and training datasets

In machine learning, there is a phenomenon called "overfitting". This occurs when models, during training, become too complex – they become really well adapted to predict the training data, but when they are asked to predict something based on new data that they haven't "seen" before, they perform poorly. In other words, the models don't *generalise* very well. To make sure that we are not creating models which are too complex, it is common practice to split the dataset into a *training* set and a *test* set. The training set is, obviously, the data that the model will be trained on, and the test set is the data that the model will be tested on after it has been trained. The amount of training data is always more numerous than the testing data, and is usually between 60-80% of the total dataset.

Again, scikit learn makes this splitting of the data into training and testing sets easy:

```
from sklearn.model_selection import train_test_split
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

In this case, we've made the test set to be 40% of the total data, leaving 60% to train with. The train_test_split function in scikit learn pushes the data randomly into the different datasets – in other words, it doesn't take the first 60% of rows as the training set and the second 40% of rows as the test set. This avoids data collection artefacts from degrading the performance of the model.

1.5.3 Setting up the output layer

As you would have been able to gather, we need the output layer to predict whether the digit represented by the input pixels is between 0 and 9. Therefore, a sensible neural network architecture would be to have an output layer of 10 nodes, with each of these nodes representing a digit from 0 to 9. We want to train the network so that when, say, an image of the digit "5" is presented to the neural network, the node in the output layer representing 5 has the highest value. Ideally, we would want to see an output looking like this: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]. However, in reality, we can settle for something like this: [0.01, 0.1, 0.2, 0.05, 0.3, 0.8, 0.4, 0.03, 0.25, 0.02]. In this case, we can take the maximum index of the output array and call that our predicted digit.

For the MNIST data supplied in the scikit learn dataset, the "targets" or the classification of the handwritten digits is in the form of a single number. We need to convert that single number into a vector so that it lines up with our 10 node output layer. In other words, if the target value in the dataset is "1" we want to convert it into the vector: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]. The code below does just that:

```
import numpy as np
def convert_y_to_vect(y):
    y_vect = np.zeros((len(y), 10))
    for i in range(len(y)):
        y_vect[i, y[i]] = 1
    return y_vect
```

```
y_v_train = convert_y_to_vect(y_train)
y_v_test = convert_y_to_vect(y_test)
y_train[0], y_v_train[0]
Out[8]:
(1, array([ 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

As can be observed above, the MNIST target (1) has been converted into the vector [0, 1, 0, 0, 0, 0, 0, 0, 0, 0], which is what we want.

1.5.4 Creating the neural network

The next step is to specify the structure of the neural network. For the input layer, we know we need 64 nodes to cover the 64 pixels in the image. As discussed, we need 10

output layer nodes to predict the digits. We'll also need a hidden layer in our network to allow for the complexity of the task. Usually, the number of hidden layer nodes is somewhere between the number of input layers and the number of output layers. Let's define a simple Python list that designates the structure of our network:

```
nn_structure = [64, 30, 10]
```

We'll use sigmoid activation functions again, so let's setup the sigmoid function and its derivative:

```
def f(x):
    return 1 / (1 + np.exp(-x))
```

```
def f_deriv(x):
    return f(x) * (1 - f(x))
```

Ok, so we now have an idea of what our neural network will look like. How do we train it? Remember the algorithm from Section 1.4.9 The final gradient descent algorithm, which we'll repeat here for ease of access and review:

Randomly initialise the weights for each layer $W^{(l)}$
While iterations < iteration limit:
1. Set ΔW and Δb to zero
2. For samples 1 to m:
a. Perform a feed forward pass through all the n_l layers. Store the activation function outputs $h^{(l)}$
b. Calculate the $\delta^{(n_l)}$ value for the output layer
c. Use backpropagation to calculate the $\delta^{(l)}$ values for layers 2 to $n_l - 1$
d. Update the $\Delta W^{(l)}$ and $\Delta b^{(l)}$ each layer
3. Perform a gradient descent step using:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)}\right]$$

$$b^{(l)} = b^{(l)} - \alpha [\frac{1}{m} \Delta b^{(l)}]$$

So the first step is to initialise the weights for each layer. To make it easy to organise the various layers, we'll use Python dictionary objects (initialised by {}). Finally, the weights have to be initialised with random values – this is to ensure that the neural network will converge correctly during training. We use the numpy library random_sample function to do this. The weight initialisation code is shown below:

```
import numpy.random as r
def setup_and_init_weights(nn_structure):
    W = {}
    b = {}
    for l in range(1, len(nn_structure)):
        W[l] = r.random_sample((nn_structure[l], nn_structure[l-1]))
        b[l] = r.random_sample((nn_structure[l],))
    return W, b
```

The next step is to set the mean accumulation values ΔW and Δb to zero (they need to be the same size as the weight and bias matrices):

```
def init_tri_values(nn_structure):
    tri_W = {}
    tri_b = {}
    for l in range(1, len(nn_structure)):
        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```

If we now step into the gradient descent loop, the first step is to perform a feed forward pass through the network. The code below is a variation on the feed forward function created in Section 1.3.5 Matrix multiplication:

```
def feed_forward(x, W, b):
    h = {1: x}
    z = {}
    for l in range(1, len(W) + 1):
        # if it is the first layer, then the input into the weights is x, otherwise,
        # it is the output from the last layer
        if l == 1:
            node_in = x
        else:
            node_in = h[l]
        z[l+1] = W[l].dot(node_in) + b[l] # z^(l+1) = W^(l)*h^(l) + b^(l)
        h[l+1] = f(z[l+1]) # h^(l) = f(z^(l))
    return h, z
```

Finally, we have to then calculate the output layer delta $\delta^{(n_l)}$ and any hidden layer delta values $\delta^{(l)}$ to perform the backpropagation pass:

```
def calculate_out_layer_delta(y, h_out, z_out):
    # delta^(nl) = -(y_i - h_i^(nl)) * f'(z_i^(nl))
    return -(y-h_out) * f_deriv(z_out)
```

```
def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    # delta^(l) = (transpose(W^(l)) * delta^(l+1)) * f'(z^(l))
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

Now we can put all the steps together into the final function:

```
def train_nn(nn_structure, X, y, iter_num=3000, alpha=0.25):
    W, b = setup_and_init_weights(nn_structure)
    cnt = 0
    m = len(y)
    avg_cost_func = []
    print('Starting gradient descent for {} iterations'.format(iter_num))
    while cnt < iter_num:
        if cnt%1000 == 0:
            print('Iteration {} of {}'.format(cnt, iter_num))
        tri_W, tri_b = init_tri_values(nn_structure)
        avg_cost = 0
        for i in range(len(y)):
            delta = {}
            # perform the feed forward pass and return the stored h and z values, to
be used in the
            # gradient descent step
            h, z = feed_forward(X[i, :], W, b)
            # loop from nl-1 to 1 backpropagating the errors
            for l in range(len(nn_structure), 0, -1):
                if l == len(nn_structure):
                    delta[l] = calculate_out_layer_delta(y[i,:], h[l], z[l])
                    avg_cost += np.linalg.norm((y[i,:]-h[l]))
                else:
                    if l > 1:
                        delta[l] = calculate_hidden_delta(delta[l+1], W[l], z[l])
                    # triW^(l) = triW^(l) + delta^(l+1) * transpose(h^(l))
                    tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.transpose(h[l]
[:,np.newaxis]))
                    # trib^(l) = trib^(l) + delta^(l+1)
                    tri_b[l] += delta[l+1]
        # perform the gradient descent step for the weights in each layer
        for l in range(len(nn_structure) - 1, 0, -1):
            W[l] += -alpha * (1.0/m * tri_W[l])
            b[l] += -alpha * (1.0/m * tri_b[l])
        # complete the average cost calculation
        avg_cost = 1.0/m * avg_cost
        avg_cost_func.append(avg_cost)
        cnt += 1
    return W, b, avg_cost_func
```

The function above deserves a bit of explanation. First, we aren't setting a termination of the gradient descent process based on some change or precision of the cost function. Rather, we are just running it for a set number of iterations (3,000 in this case) and we'll

monitor how the average cost function changes as we progress through the training (avg_cost_func list in the above code). In each iteration of the gradient descent, we cycle through each training sample (range(len(y)) and perform the feed forward pass and then the backpropagation. The backpropagation step is an iteration through the layers starting at the output layer and working backwards – range(len(nn_structure), 0, -1). We calculate the average cost, which we are tracking during the training, at the output layer (l == len(nn_structure)). We also update the mean accumulation values, ΔW and Δb, designated as tri_W and tri_b, for every layer apart from the output layer (there are no weights connecting the output layer to any further layer).

Finally, after we have looped through all the training samples, accumulating the tri_W and tri_b values, we perform a gradient descent step change in the weight and bias values:

$$W^{(l)} = W^{(l)} - \alpha[\frac{1}{m}\Delta W^{(l)}]$$

$$b^{(l)} = b^{(l)} - \alpha[\frac{1}{m}\Delta b^{(l)}]$$

After the process is completed, we return the trained weight and bias values, along with our tracked average cost for each iteration. Now it's time to run the function – NOTE: this may take a few minutes depending on the capabilities of your computer.

```
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train)
```

Now we can have a look at how the average cost function decreased as we went through the gradient descent iterations of the training, slowly converging on a minimum in the function:

```
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```

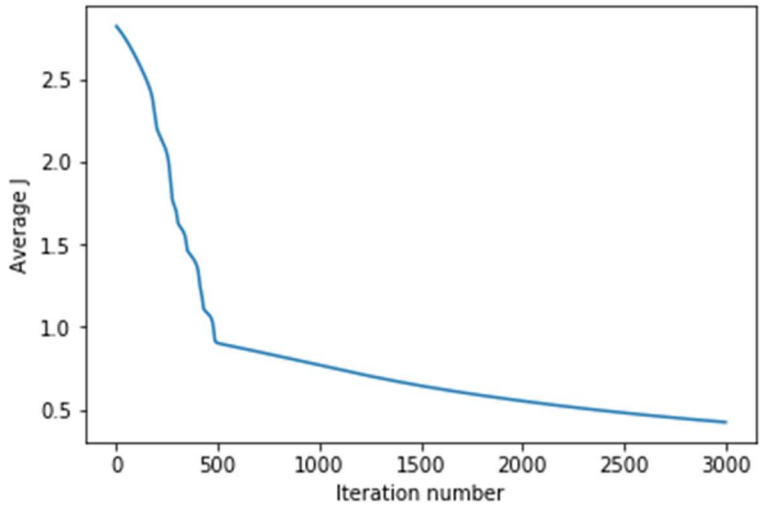

Figure 14 Average J versus the number of iterations

We can see in the above plot, that by 3,000 iterations of our gradient descent our average cost function value has started to "plateau" and therefore any further increases in the number of iterations isn't likely to improve the performance of the network by much.

1.5.5 Assessing the accuracy of the trained model

Now that we've trained our MNIST neural network, we want to see how it performs on the test set. Is our model any good? Given a test input (64 pixels), we need to find what the output of our neural network is – we do that by simply performing a feed forward pass through the network using our trained weight and bias values. As discussed previously, we assess the prediction of the output layer by taking the node with the maximum output as the predicted digit. We can use the numpy.argmax function for this, which returns the index of the array value with the highest value:

```
def predict_y(W, b, X, n_layers):
    m = X.shape[0]
    y = np.zeros((m,))
    for i in range(m):
        h, z = feed_forward(X[i, :], W, b)
        y[i] = np.argmax(h[n_layers])
    return y
```

Finally, we can assess the accuracy of the prediction (i.e. the percentage of times the network predicted the handwritten digit correctly), by using the scikit learn accuracy_score function:

```
from sklearn.metrics import accuracy_score
y_pred = predict_y(W, b, X_test, 3)
accuracy_score(y_test, y_pred)*100
```

This gives an 86% accuracy of predicting the digits. Sounds pretty good right? Well actually no, it's pretty bad. The current state-of-the-art deep learning algorithms achieve accuracy scores of 99.7% (see here), so we are a fair way off that sort of accuracy.

There are many more exciting things to learn. As I said at the beginning, this book is simply "Part A" of a much more comprehensive book "From 0 to TensorFlow" (forthcoming) which will cover the implementation of deep learning architectures such as convolutional neural networks, recurrent neural networks, visualisation and other tips and tricks in the exciting new TensorFlow framework. Stay tuned for further news on the release of this book.

As always, have fun!